#Coding
VentiVentisei

Dependency Injection in .NET: principi, design e anti-pattern

Alessandro Mengoli

Microsoft


AGIC


present
exprivia group


BizzyNow™
make yourself better

```
public class UserService
{
    public void CreateUser(User user)
    {
        /* some business logic here */

        var storage = new DBStorage();
        storage.Save(user);
    }
}
```

```
public class UserServ
{
    public void Creat
    {
        /* some busin

        var storage =
        storage.Save(
    }
}
```

Perché è sbagliato

UserService deve conoscere solo logica di business

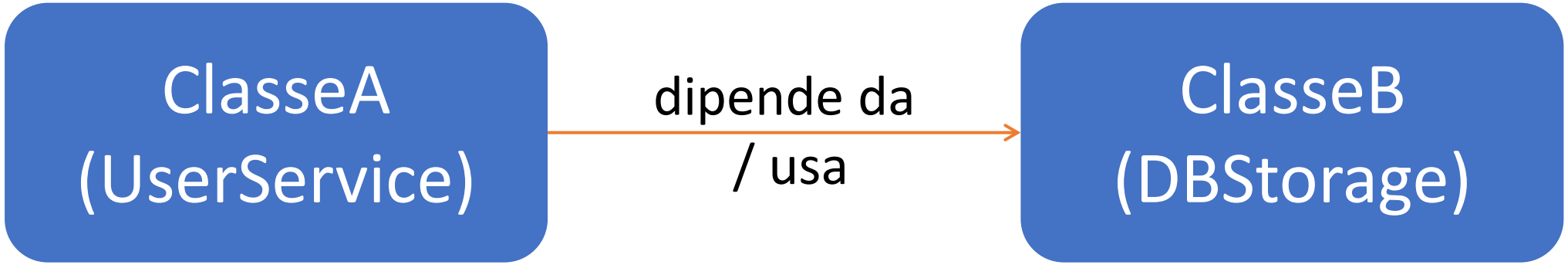
Se cambio tecnologia devo cambiare ovunque

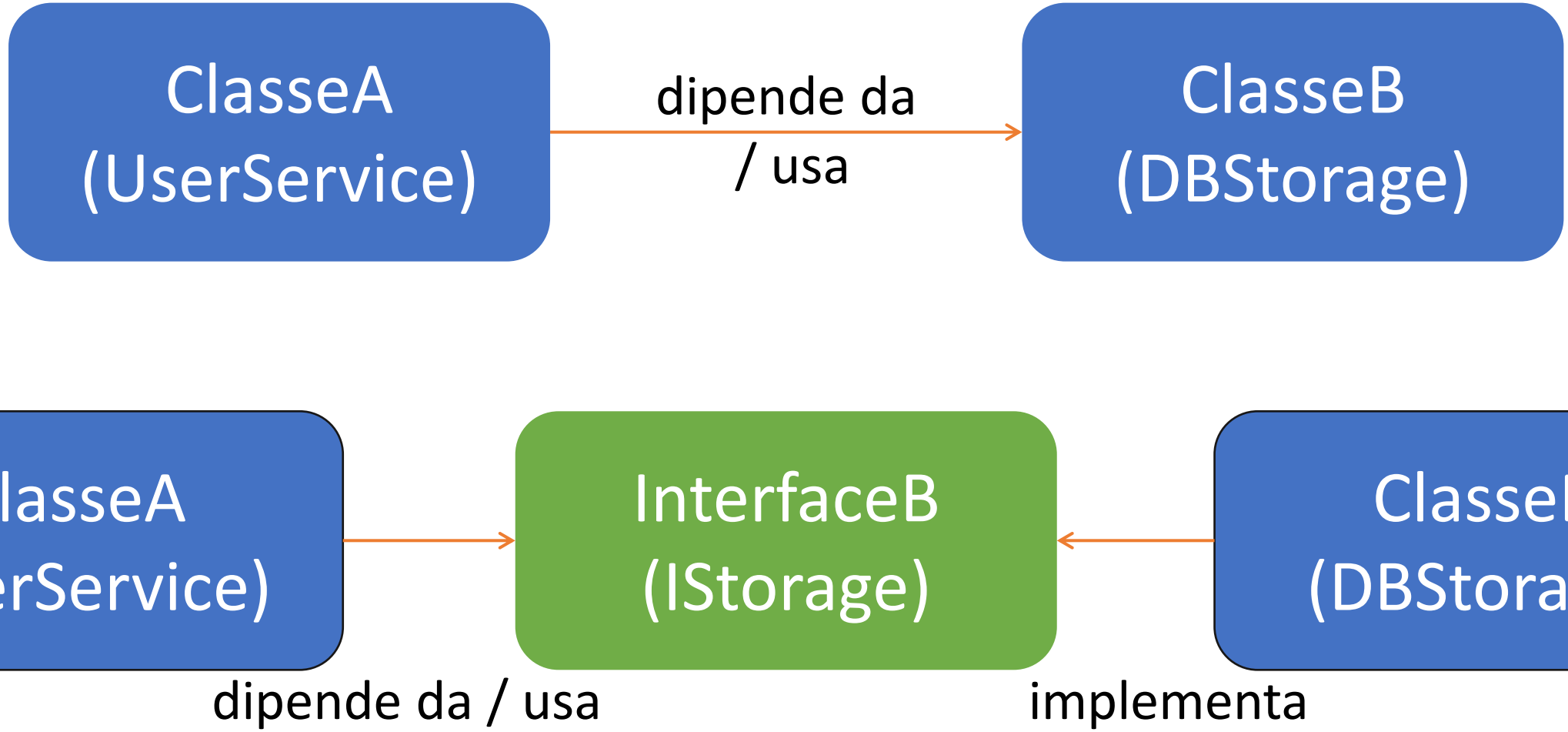
Non posso fare test unitari!

Non posso riutilizzare la logica

Troppo coeso, troppe responsabilità

- **S**RP - Single Responsibility Principle
- **O**PC - Open/Closed Principle
- **L**SP - Liskov Substitution Principle
- **I**SP - Interface Segregation Principle
- **D**IP - Dependency Inversion Principle


ClasseA
(UserService)
dipende da
/ usa
ClasseB
(DBStorage)


ClasseA
(UserService)
dipende da
/ usa
ClasseB
(DBStorage)
ClasseA
(UserService)
InterfaceB
(IStorage)
ClasseB
(DBStorage)
dipende da / usa
implementa

```
public class UserService(IStorage storage)
{
    public void CreateUser(User user)
    {
        /* some business logic here */

        storage.Save(user);
    }
}
```

DI .NET

```
public class MyClass(INotifier notifier){

    public void MyMethod(){
        … do something ….
        notifier.Notify();
     }

}
```

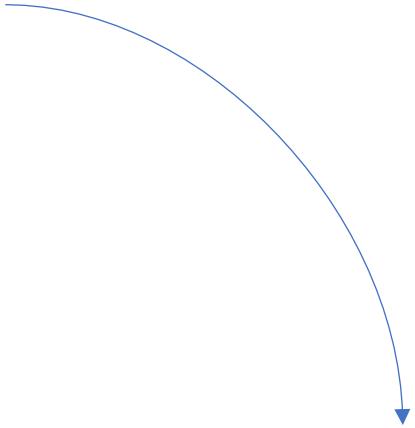
DI .NET

```
public class MyClass(INotifier notifier){

    public void MyMethod(){
        … do something ….
        notifier.Notify();
     }

}
```

```
builder.Services.AddScoped<MailNotifier, INotifier>();
```



DI Container

È un registry che:

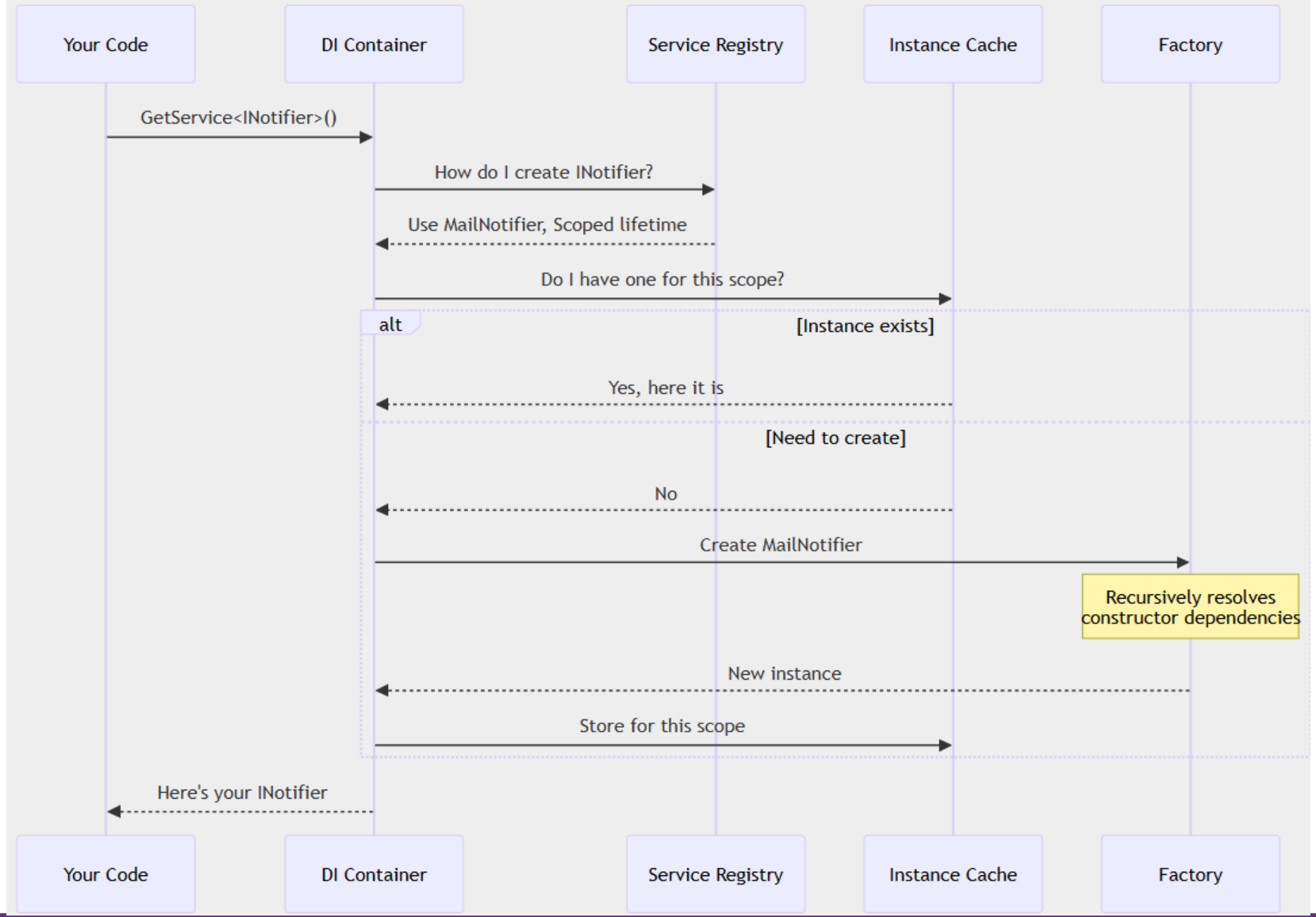
- Conosce i servizi esistenti (interfacce/astrazioni)
- Sa come creare i servizi (implementazioni concrete)
- Quando creare una nuova istanza o riutilizza (lifetime)

DI Container – Add*

```
builder.Services.AddScoped<MailNotifier, INotifier>();
```

Aggiungi una entry al registry che dice:

«quando qualcuno chiede un INotifier allora dagli un MailNotifier e come lifetime scoped»


Your Code
DI Container
Service Registry
Instance Cache
Factory
GetService<INotifier>()
How do I create INotifier?
Use MailNotifier, Scoped lifetime
Do I have one for this scope?
alt
[Instance exists]
Yes, here it is
[Need to create]
No
Create MailNotifier
Recursively resolves constructor dependencies
New instance
Store for this scope
Here's your INotifier
Your Code
DI Container
Service Registry
Instance Cache
Factory

Lifetime

SINGLETON

Una singola istanza per tutta la durata dell'applicazione

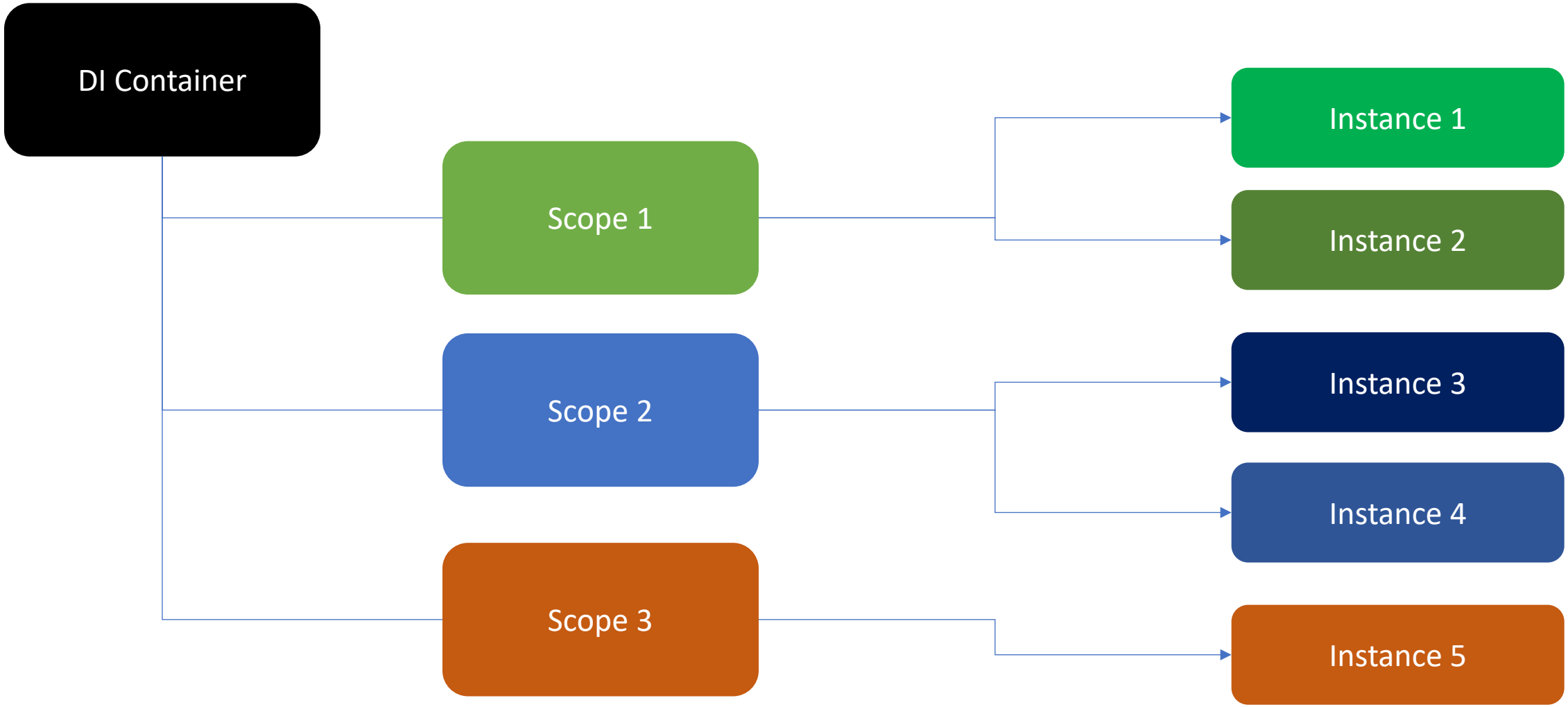
SCOPED

Una istanza per ogni scope

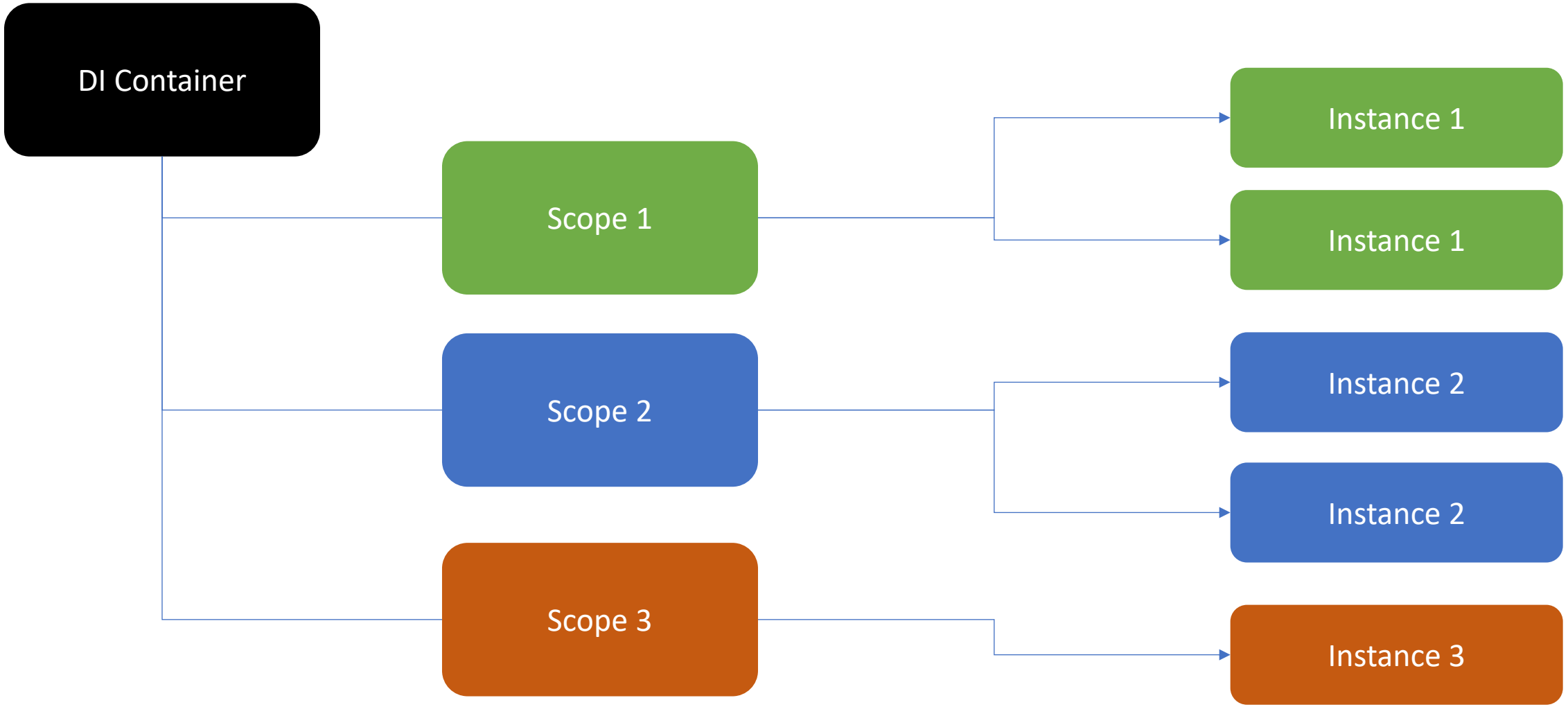
TRANSIENT

Una nuova istanza ogni volta che viene richiesto

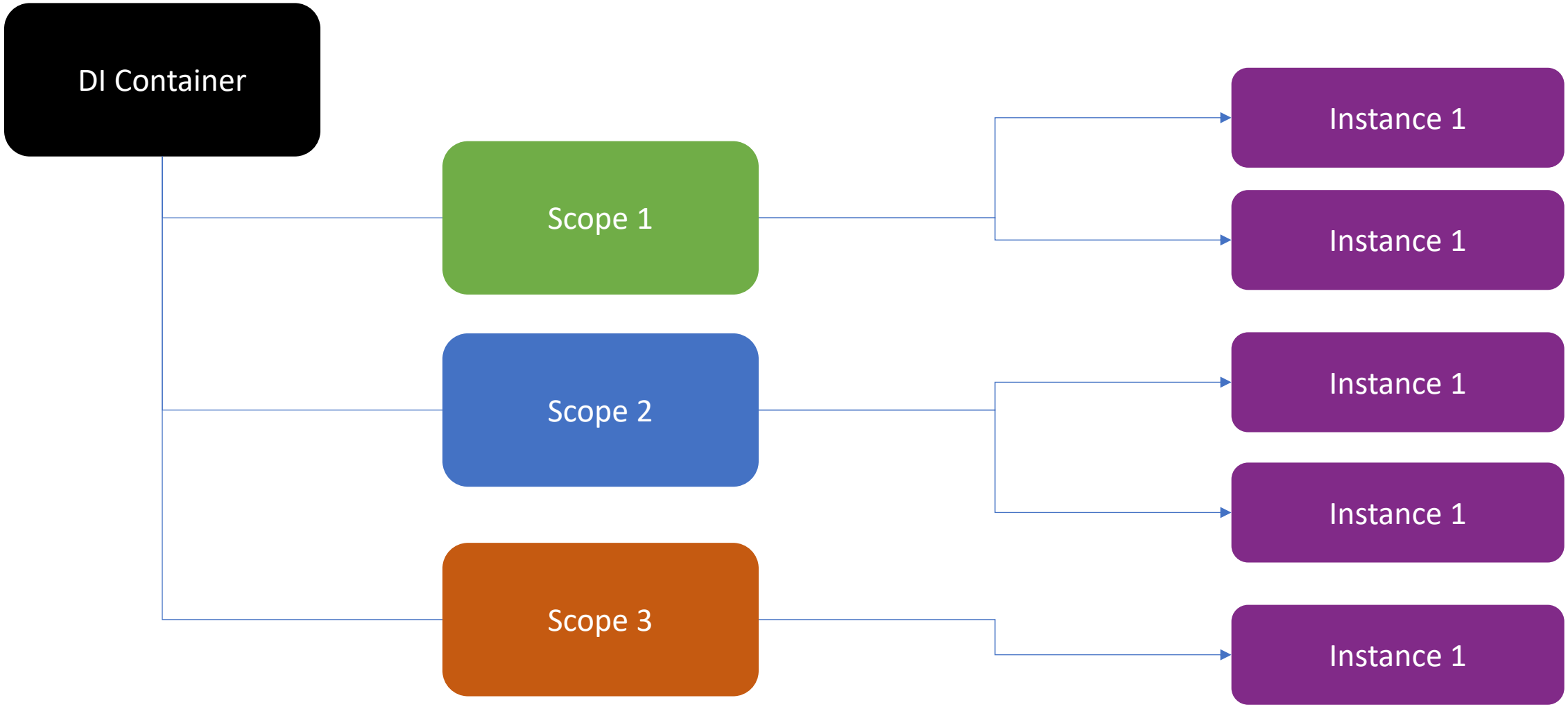
Transient


DI Container
Scope 1
Scope 2
Scope 3
Instance 1
Instance 2
Instance 3
Instance 4
Instance 5

Scoped


DI Container
Scope 1
Scope 2
Scope 3
Instance 1
Instance 1
Instance 2
Instance 2
Instance 3

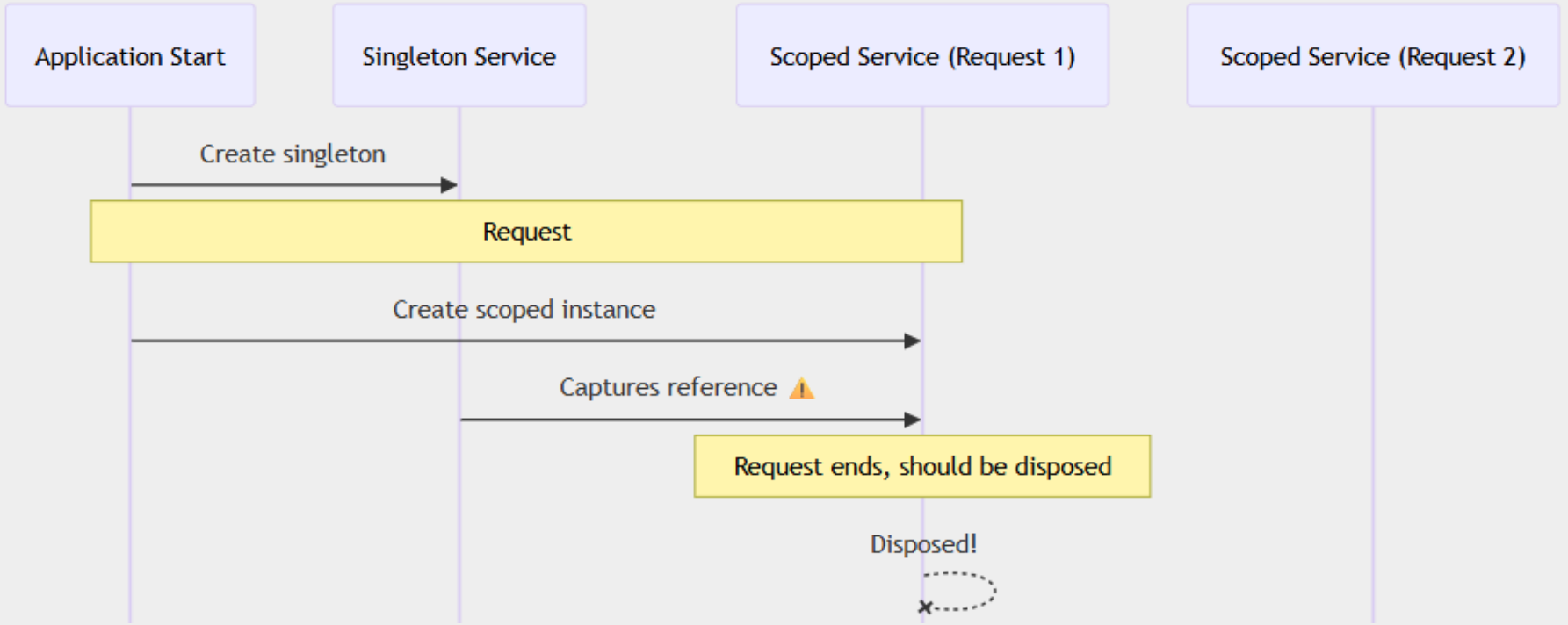
Singleton

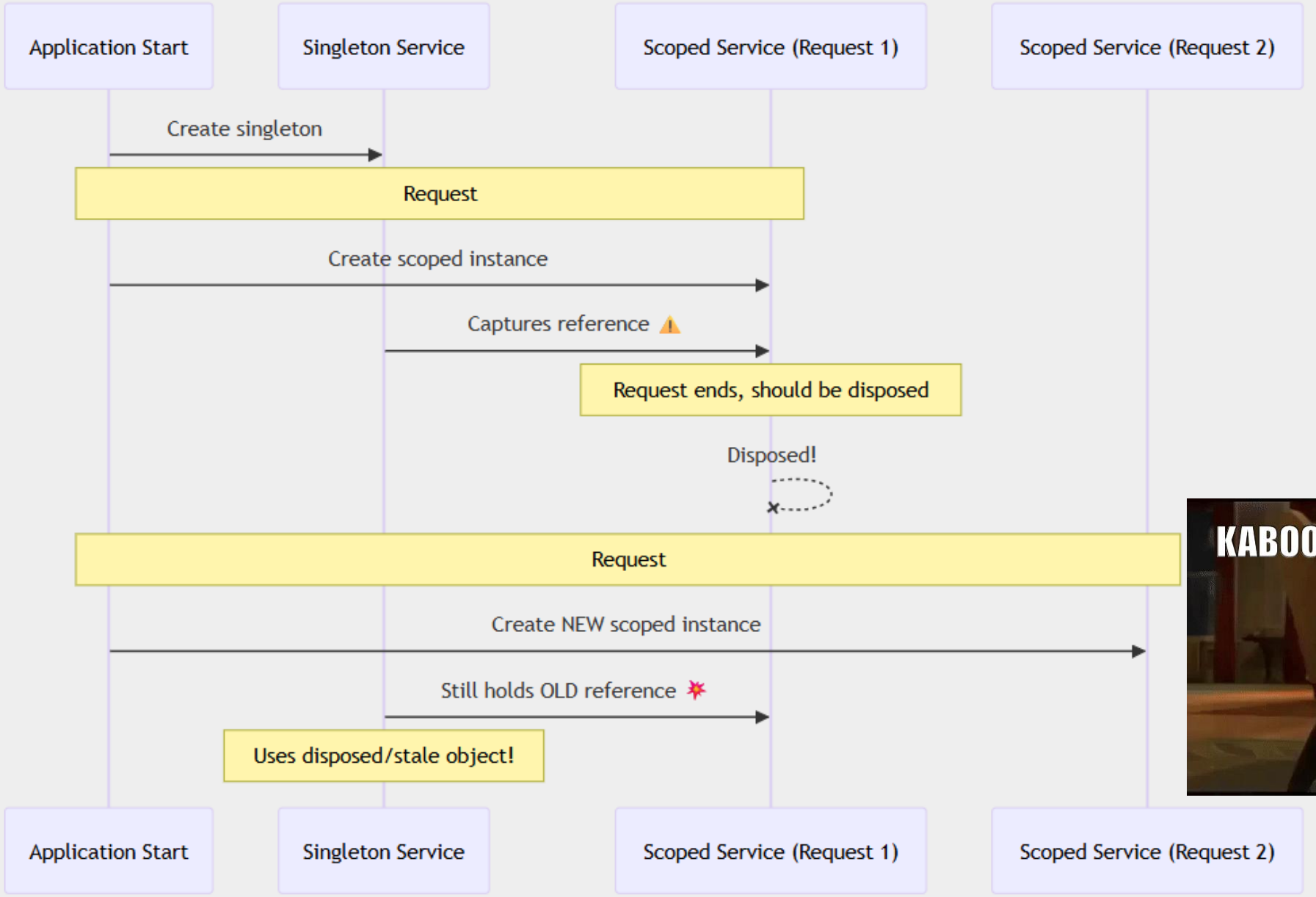

DI Container
Scope 1
Scope 2
Scope 3
Instance 1
Instance 1
Instance 1
Instance 1
Instance 1

Occhio!

Fare con attenzione la corretta gestione gerarchia dipendenze

Un service deve dipendere da un service con lifetime uguale o superiore


Application Start
Singleton Service
Scoped Service (Request 1)
Scoped Service (Request 2)
Create singleton
Request
Create scoped instance
Captures reference
Request ends, should be disposed
Disposed!


Application Start
Singleton Service
Scoped Service (Request 1)
Scoped Service (Request 2)
Create singleton
Request
Create scoped instance
Captures reference
Request ends, should be disposed
Disposed!
Request
Create NEW scoped instance
Still holds OLD reference
Uses disposed/stale object!
Application Start
Singleton Service
Scoped Service (Request 1)
Scoped Service (Request 2)


KABOOM?..YES RICO KABOOM..
makeagif.com

Più implementazioni stessa Interfaccia

```
services.AddSingleton<INotificationService, EmailService>();
services.AddSingleton<INotificationService, SmsService>();
services.AddSingleton<INotificationService, PushService>();
```

Più implementazioni stessa Interfaccia

```
services.AddSingleton<INotificationService, EmailService>();
services.AddSingleton<INotificationService, SmsService>();
services.AddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service)
```

Più implementazioni stessa Interfaccia

```
services.AddSingleton<INotificationService, EmailService>();
services.AddSingleton<INotificationService, SmsService>();
services.AddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service) // Riceve PushService
```

Più implementazioni stessa Interfaccia

```
services.AddSingleton<INotificationService, EmailService>();
services.AddSingleton<INotificationService, SmsService>();
services.AddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service) // Riceve PushService

public MyClass(IEnumerable<INotificationService> services)
```

Più implementazioni stessa Interfaccia

```
services.AddSingleton<INotificationService, EmailService>();
services.AddSingleton<INotificationService, SmsService>();
services.AddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service) // Riceve PushService

public MyClass(IEnumerable<INotificationService> services) // Riceve
[EmailService, SmsService, PushService]
```

Più implementazioni stessa Interfaccia

```
services.TryAddSingleton<INotificationService, EmailService>();
services.TryAddSingleton<INotificationService, SmsService>();
services.TryAddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service)
```

Più implementazioni stessa Interfaccia

```
services.TryAddSingleton<INotificationService, EmailService>();
services.TryAddSingleton<INotificationService, SmsService>();
services.TryAddSingleton<INotificationService, PushService>();
```

```
public MyClass(INotificationService service) // Riceve EmailService
```

Keyed Services (da .NET8)

I **KeyedServices** in .NET permettono di registrare e risolvere **più implementazioni della stessa interfaccia**, distinguendole tramite una **chiave** (string, enum, object).

```
// Registrazione con chiave
builder.Services.AddKeyedSingleton<INotificationService, EmailService>("email")
builder.Services.AddKeyedSingleton<INotificationService, SmsService>("sms");
```

Keyed Services (da .NET8)

```
// Registrazione con chiave
builder.Services.AddKeyedSingleton<INotificationService, EmailService>("email");
builder.Services.AddKeyedSingleton<INotificationService, SmsService>("sms");


// Risoluzione tramite chiave
public class NotificationController{

public NotificationController(
  [FromKeyedServices("email")] INotificationService emailService,
[FromKeyedServices("sms")] INotificationService smsService)

   {        // Usa il servizio specifico    }

}
```

Anti-pattern

Captive dependency

Transient disposable

Service Locator

Async DI Factory

Breaking change

About & resources

- Usate la DI e rispettate i principi SOLID
- Abilitate la validazione scope
- Sfruttate Copilot (o similare) per fare dei check, magari con MCP

About & resources

amengoli9/talks/

devpills.net

devromagna.org

Alessandro Mengoli

Technical Leader @ SACMI